

Neural Networks

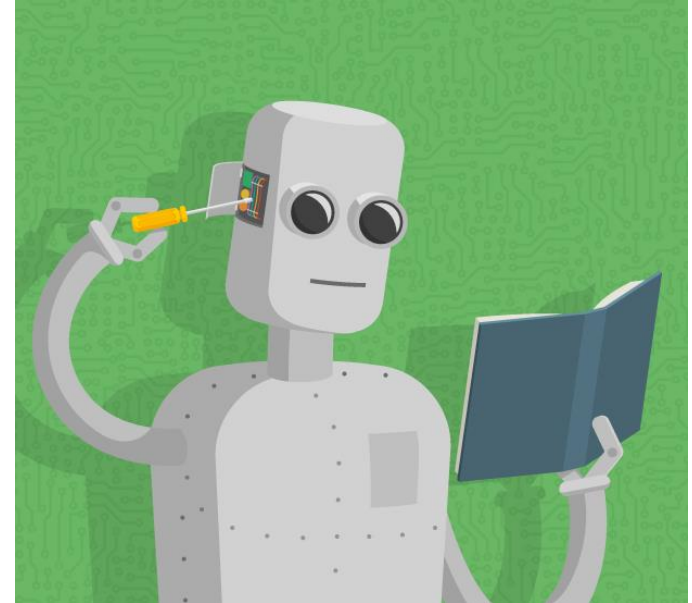
Scalable Data Engineering

Agenda

- What is machine learning?
- Terms and concepts: neuron
- Perceptron as logical classifier
- Perceptron training
- Sigmoid neurons
- Gradient descent
- Training with backpropagation
- Activation functions
- Regularization

Definition

- Term covers a lot...
- An artificial system „learns“ patterns and rules from a *training set*
- Learned knowledge is applied to evaluate unknown data
- Close relation with computational statistics / data mining
- Applications: fraud detection, stock analysis, Voice recognition, spam filter,



Supervised Learning

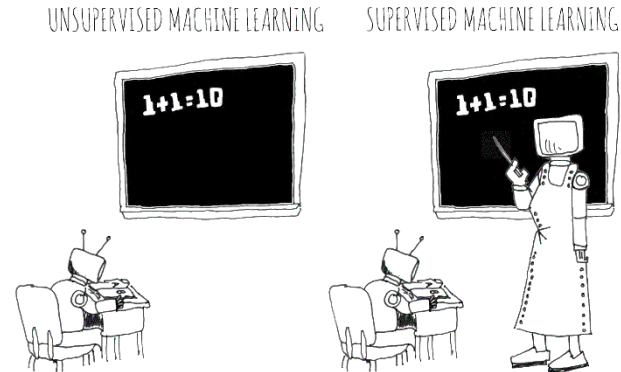
- Machine learns based on example inputs and desired outputs provided by a „teacher“
- Goal: learn general rules for mapping input to output

Unsupervised Learning

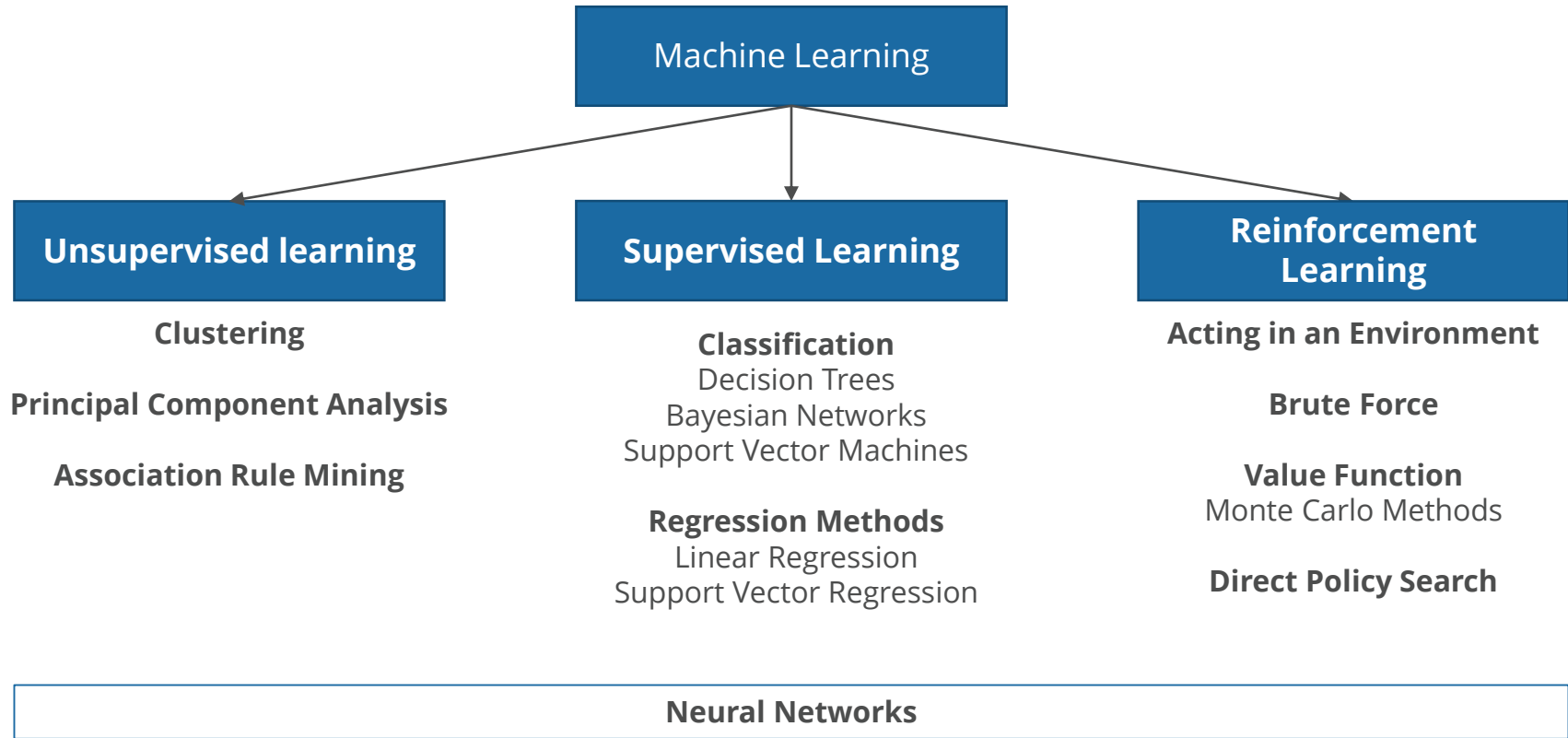
- Machine is left alone for learning and tries to find rules / structures independently in its input
- Goal: create a model / rules that allows description

Reinforcement Learning

- Machine interacts with a dynamic environment with rewards and punishments
- Goal: learn what to do in a certain situation to achieve a certain goal



Overview



Supervised and Unsupervised Learning

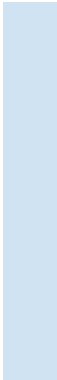
Supervised Learning

- Infer a function from labeled data

Unsupervised Learning

- Description of hidden structure in unlabeled data

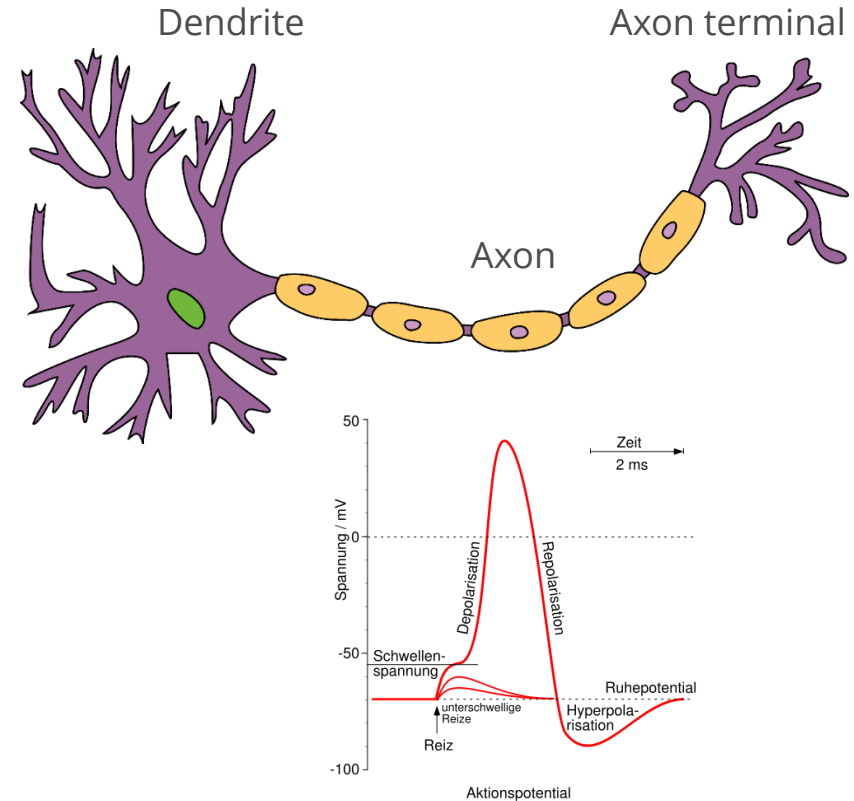
	Supervised Learning	Unsupervised Learning
Given data	▪ Set of example pairs (x, y), i.e., data x with labels y	▪ Set of data x without labels ▪ Cost function to be minimized
Goal	▪ Find $f: X \rightarrow Y$ that is implied by the data ▪ A cost function expresses the relative mismatch between y and $f(x)$	▪ Minimizing the cost function, thus comparing x with the output f
Applications	➤ Learning Neural Networks ➤ Regression ➤ Forecasting ➤ Classification	➤ Clustering ➤ Association Rule Mining ➤ Anomaly detection



Neural Networks

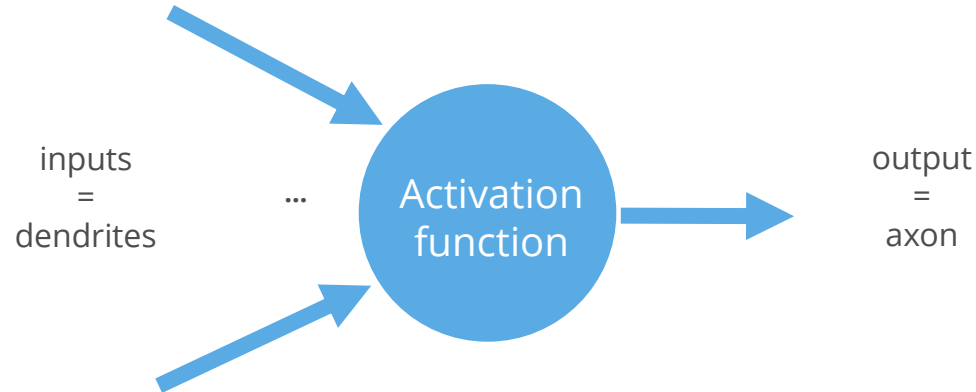
Biological Basics

- Neuron/nerve cell is the structural and functional building block of nervous system
- Cell specialises in receiving, processing and transmitting excitation
- Dendrites receive excitation from other cells
- Axon transmits excitation to other cells
- Transmission: internal electric, external chemical (neurotransmitter, synapses)
- Incoming excitation summed at axon hillock
- If threshold potential is exceeded, action potential is released (all-or-nothing)
- Analog/Digital Converter



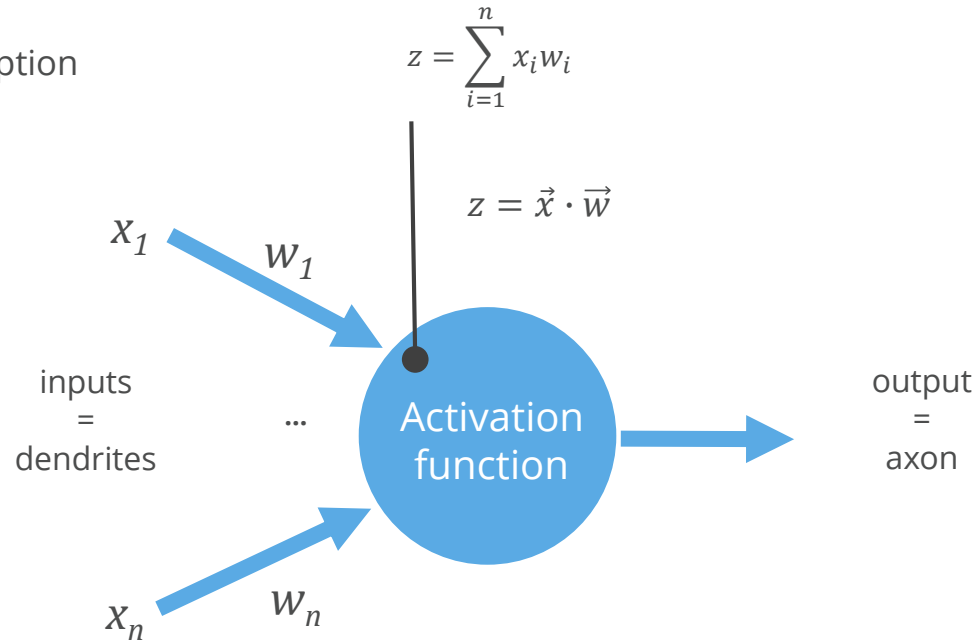
Basics

- Core element is the artificial neuron → Perceptron
- Published 1958



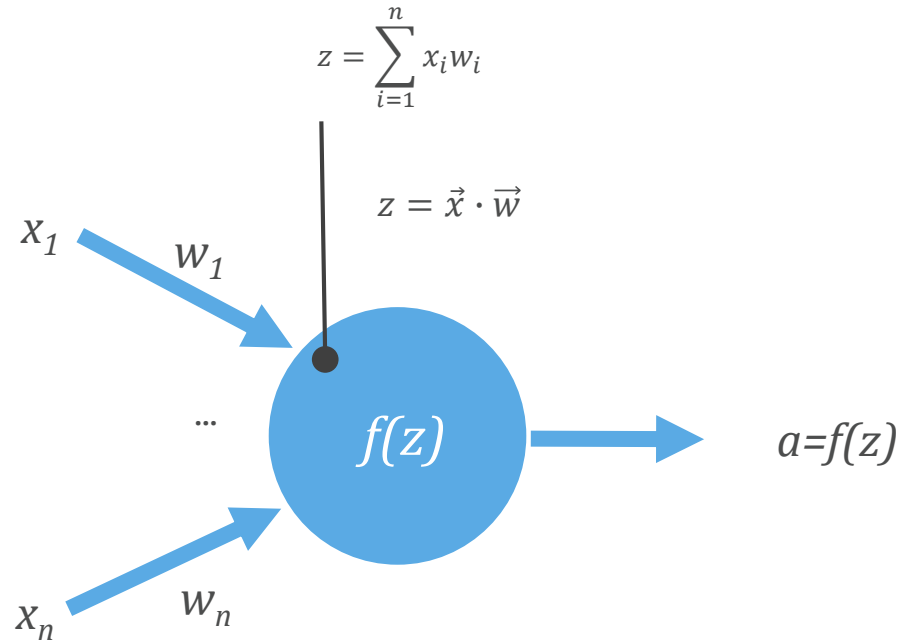
Basics

- Mathematical description



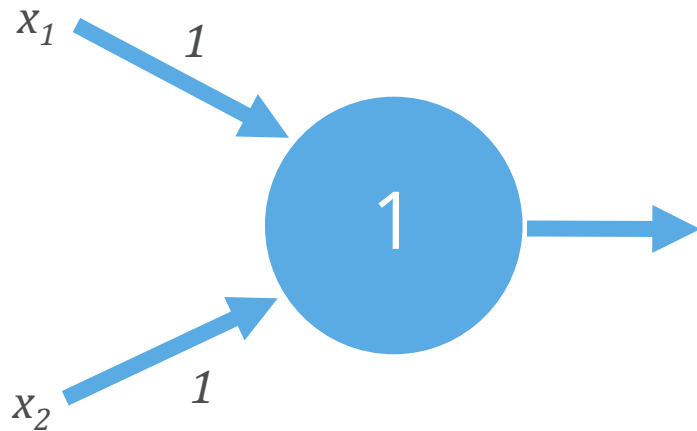
Basics

- Mathematical description

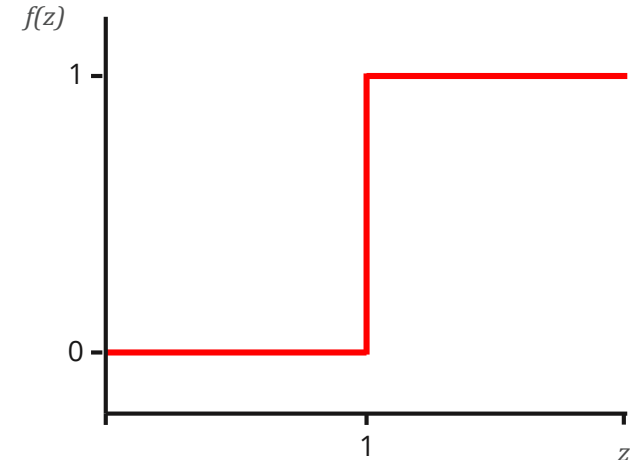


Perceptron

- Simplified neural network
- Logical threshold element, input/output boolean (true/false, 1/0)
- Activation function: threshold $\rightarrow$ rectangular function

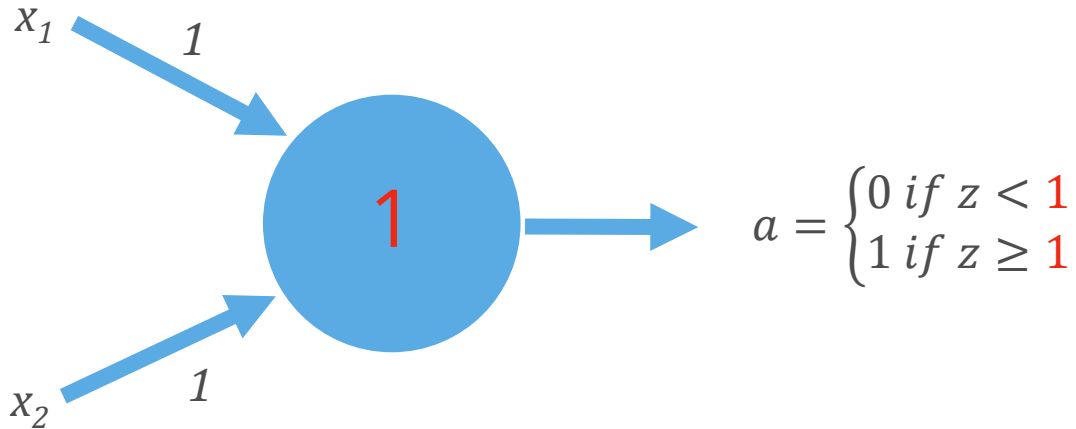


$$a = \begin{cases} 0 & \text{if } z < 1 \\ 1 & \text{if } z \geq 1 \end{cases}$$



Perceptron

- Simplified neural network
- Logical threshold element, input/output boolean (true/false, 1/0)
- Activation function: **threshold** $\rightarrow$ rectangular function

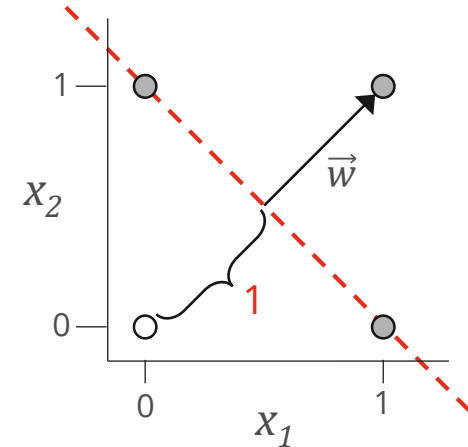
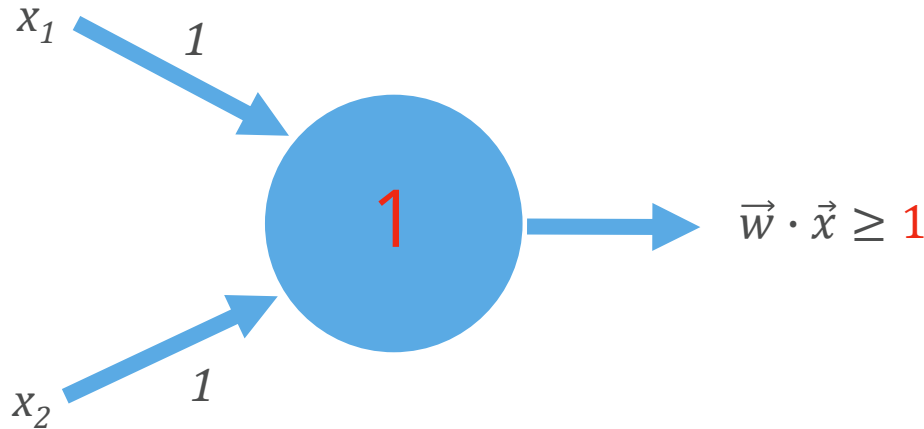


x_1	x_2	a
1	0	1
0	1	1
1	1	1
0	0	0

logical OR

Perceptron

- Simplified neural network
- Logical threshold element, input/output boolean (true/false, 1/0)
- Activation function: **threshold** $\rightarrow$ rectangular function

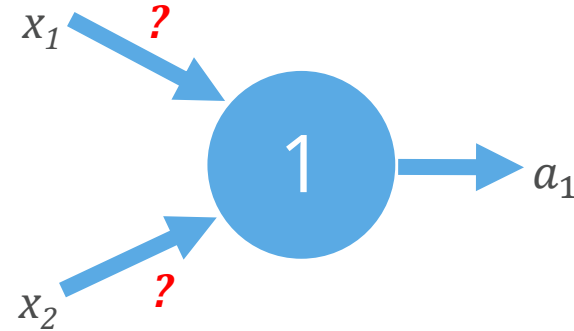


Perceptron learning algorithm

- How to learn weights?
- Reinforced learning, paths connecting active inputs and active outputs are reinforced
- Hebb's rule: „what fires together, wires together“
- $\Delta w_{ij} = \text{learning rate} \cdot \text{input}_i \cdot \text{output}_j$

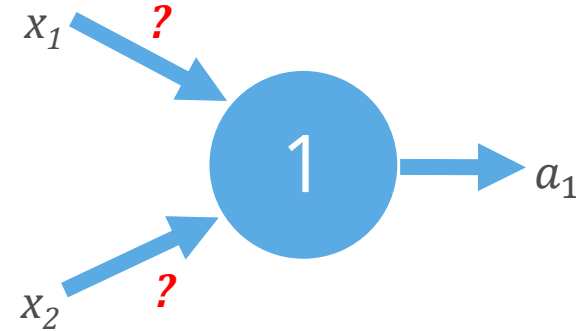
$$w_{ij}^{\text{new}} = w_{ij}^{\text{old}} + \Delta w_{ij}$$

$$\Delta w_{ij} = \alpha \cdot x_i \cdot a_j$$



Perceptron learning algorithm

- How to learn weights?
- Corrective learning, weights are adjusted according to the difference between wanted and actual output
- if $a_1 = 1$ (resp. 0) and the wanted output is 1 (resp. 0), weights are not changed
- if $a_1 = 0$, but should be 1, weights are incremented.
- If $a_1 = 1$, but should be 0, weights are decreased.
- → delta rule



Perceptron learning algorithm

- After each trainings step:

$$w_{ij}^{new} = w_{ij}^{old} + \Delta w_{ij}$$

- with

$$\Delta w_{ij} = \alpha \cdot (t_j - a_j) \cdot x_i$$

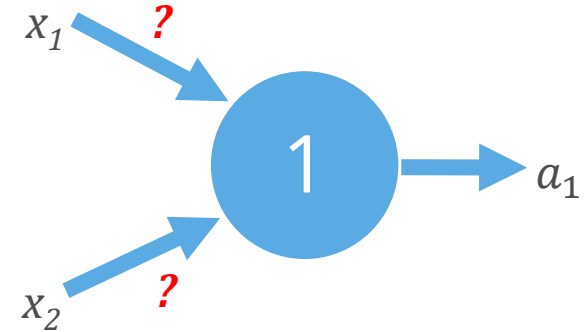
„learning rate · error · input“

- $\alpha > 0$ constant learning rate
- x_i i-th input
- t_j wanted value for j-th output neuron for input x
- a_j actual value for j-th output neuron for input x

Perceptron Training

Example: OR (Delta Rule)

1. Initialize weights (random $[-1,1]$)
2. Select input for training step
3. Calculate output
4. Update weights
5. Repeat from 2. or terminate if error is acceptable

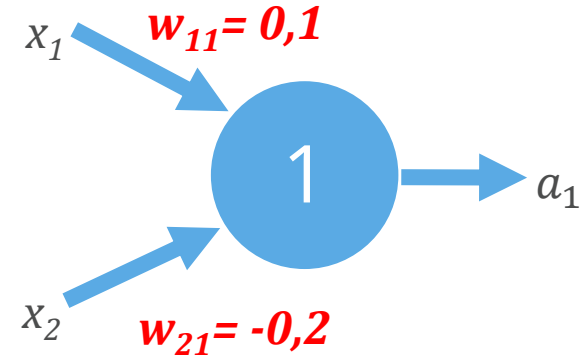


	x_1	x_2	t_1	a_1
1	1	0	1	
2	0	1	1	
3	0	0	0	
4	1	1	1	

Perceptron Training

Example: OR (Delta Rule)

1. Initialize weights (random $[-1,1]$)
2. Select input for training step
3. Calculate output
4. Update weights
5. Repeat from 2. or terminate if error is acceptable



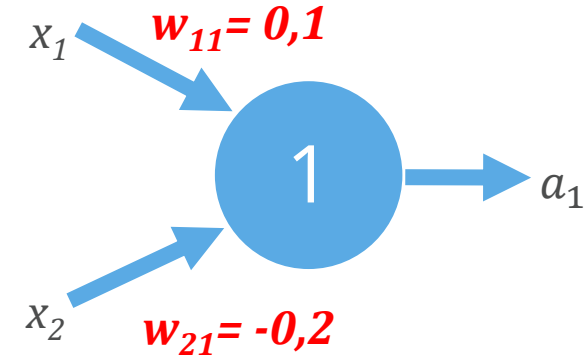
	x_1	x_2	t_1	a_1
1	1	0	1	
2	0	1	1	
3	0	0	0	
4	1	1	1	

$$\alpha = 0,3$$

Perceptron Training

Example: OR (Delta Rule)

1. Initialize weights (random $[-1,1]$)
2. Select input for training step
3. Calculate output
4. Update weights
5. Repeat from 2. or terminate if error is acceptable



	x_1	x_2	t_1	a_1
1	1	0	1	
2	0	1	1	
3	0	0	0	
4	1	1	1	

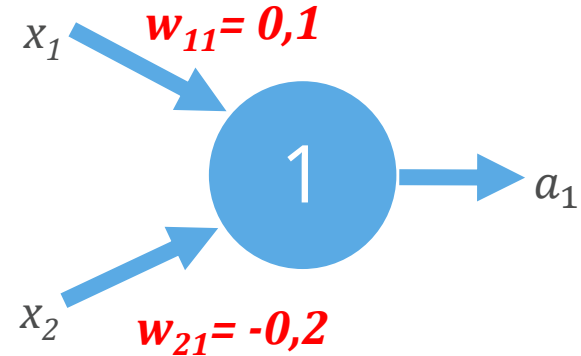
$$\alpha = 0,3$$

Example: OR (Delta Rule)

1. Initialize weights (random $[-1,1]$)
2. Select input for training step
3. Calculate output
4. Update weights
5. Repeat from 2. or terminate if error is acceptable

$$\begin{aligned}a_1 &= f((0,1 \cdot 1) + (-0,2 \cdot 0)) \\&= f(0,1) \rightarrow \text{no activation } (0,1 < 1) \\&= 0\end{aligned}$$

$$\alpha = 0,3$$



	x_1	x_2	t_1	a_1
1	1	0	1	0
2	0	1	1	
3	0	0	0	
4	1	1	1	

Example: OR (Delta Rule)

1. Initialize weights (random [-1,1])
2. Select input for training step
3. Calculate output
4. Update weights
5. Repeat from 2. or terminate if error is acceptable

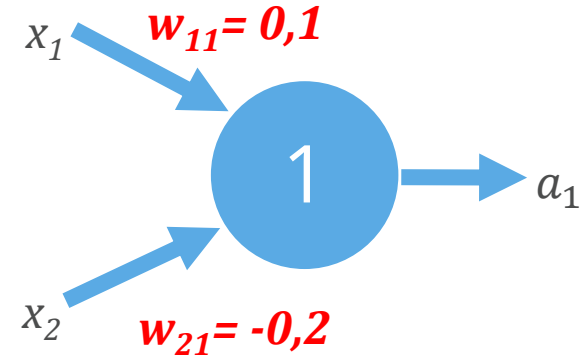
$$\Delta w_{11} = 0,3 \cdot (1 - 0) \cdot 1 = 0,3$$

$$\Delta w_{21} = 0,3 \cdot (1 - 0) \cdot 0 = 0$$

$$w'_{11} = 0,1 + 0,3 = 0,4$$

$$w'_{21} = -0,2 + 0 = -0,2$$

$$\alpha = 0,3$$

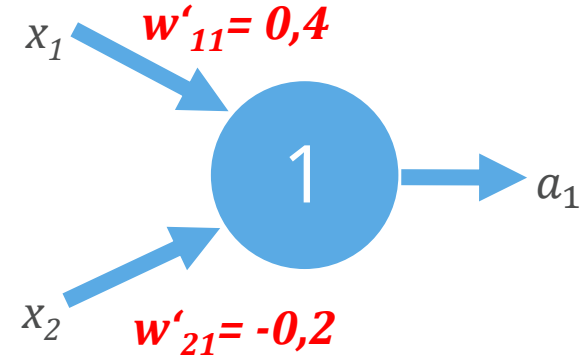


	x_1	x_2	t_1	a_1
1	1	0	1	0
2	0	1	1	
3	0	0	0	
4	1	1	1	

Perceptron Training

Example: OR (Delta Rule)

1. Initialize weights (random $[-1,1]$)
2. Select input for training step
3. Calculate output
4. Update weights
5. Repeat from 2. or terminate if error is acceptable

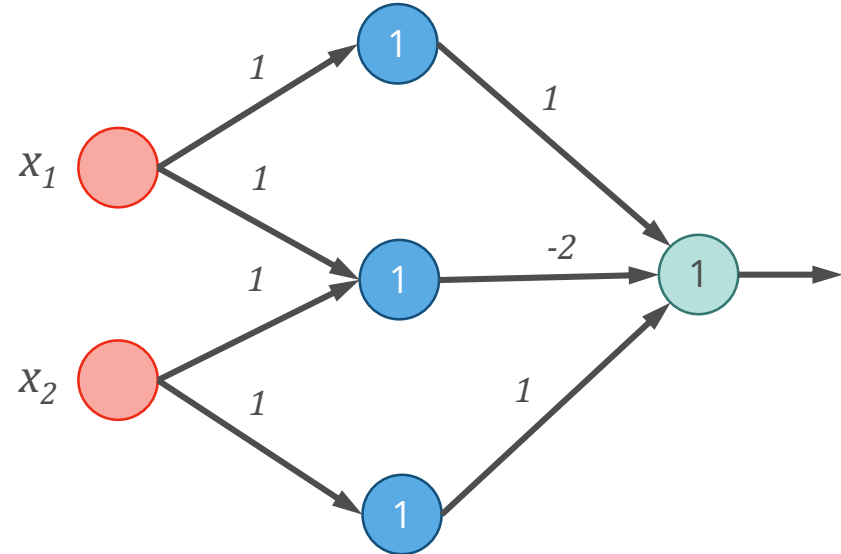
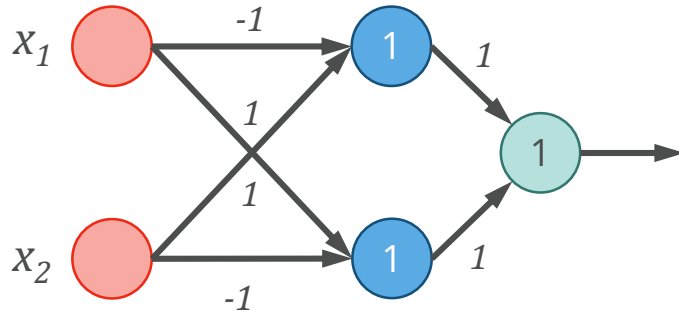


	x_1	x_2	t_1	a_1
1	1	0	1	0
2	0	1	1	
3	0	0	0	
4	1	1	1	

$$\alpha = 0,3$$

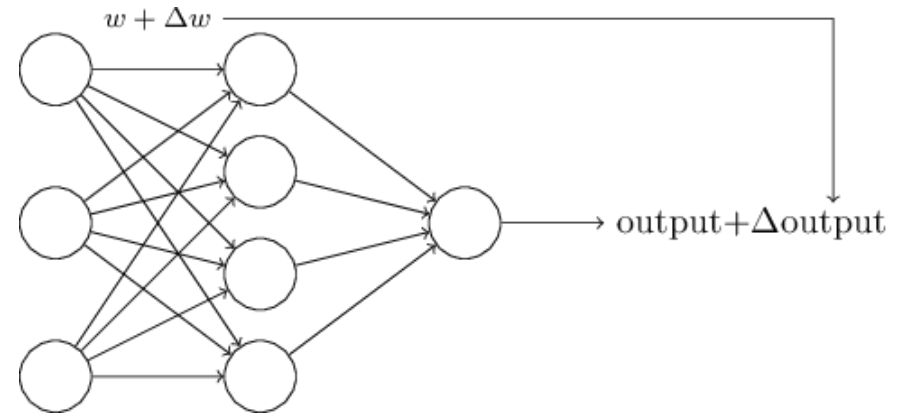
Multilayer Perceptron

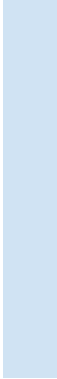
- Hidden Layer
- Calculate Features
- Arbitrary Combinations



Limits of the perceptron

- “All or Nothing” principle
- Weight updates have no impact until neuron fires
- “fine-tuning” complex networks is not possible with perceptrons
- Wanted behaviour:
 - Small change in weights
 - Small change in output
- With Perceptrons:
 - Small change in weights
 - Perceptron can “flip”
 - Cascade effect
 - Network/Output changes strongly!



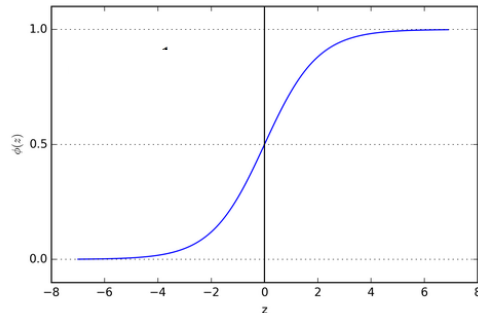


Training with Backpropagation

Another activation function

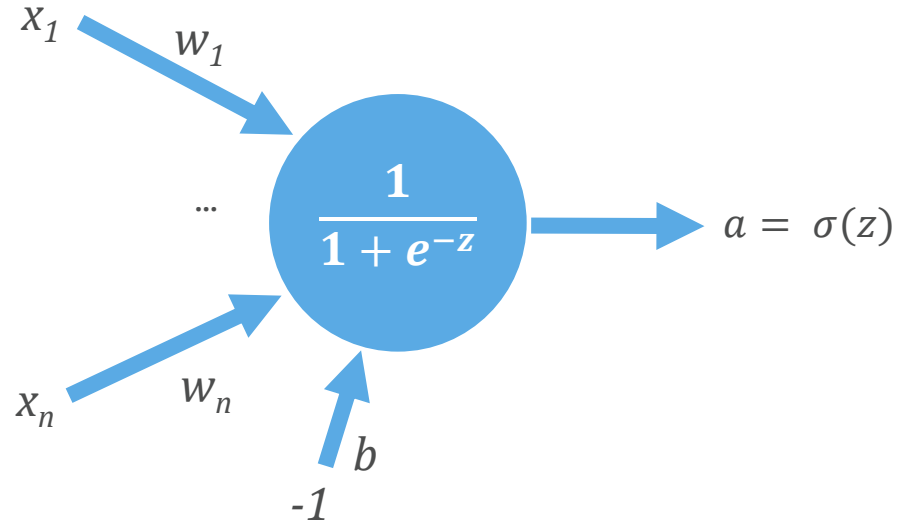
Sigmoid Neuron

- Smoothed Perceptron
- Basic structure is kept
- Inputs and outputs range from 0 to 1
- Activation thresholds are modelled as constant input / bias
- Activation function → Sigmoid function



$$z = b + \sum_{i=1}^n x_i w_i$$

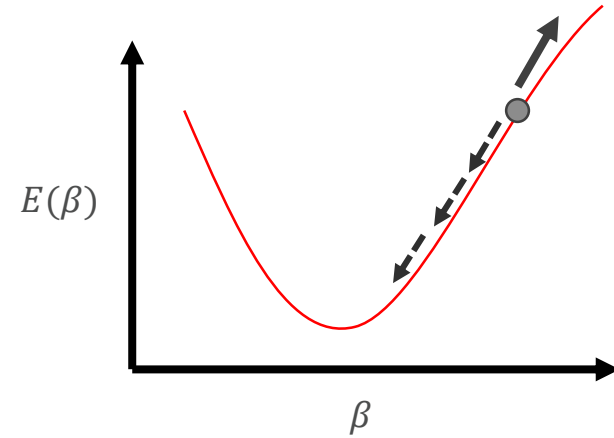
$$z = b + \vec{x} \cdot \vec{w}$$



Training: Optimization Problem

Learning algorithm

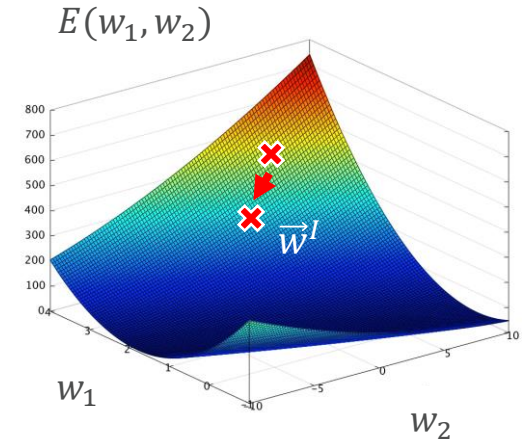
- Training weights in multi-layer networks
- Goal: minimize difference between wanted and actual output
- → minimize a cost function E
- Gradient descent
 - Start with random parameter
 - Calculate des slope of E for β
 - Move in the opposite direction to reduce error
 - Step-by-step until (local) minimum is reached



Gradient descent

- $\rightarrow$ minimize a cost function E with multiple parameters
- Example: neuron with two inputs $\rightarrow E(w_1, w_2)$
- Calculate gradient $\nabla E(w_1, w_2) \rightarrow$ points into direction of biggest increase
- Negative gradient points into direction of biggest reduction
- Gradient is a vector containing the partial derivatives of the cost function

$$\nabla E(w_1, \dots, w_n) = \left\langle \frac{\partial E}{\partial w_1}, \dots, \frac{\partial E}{\partial w_n} \right\rangle$$

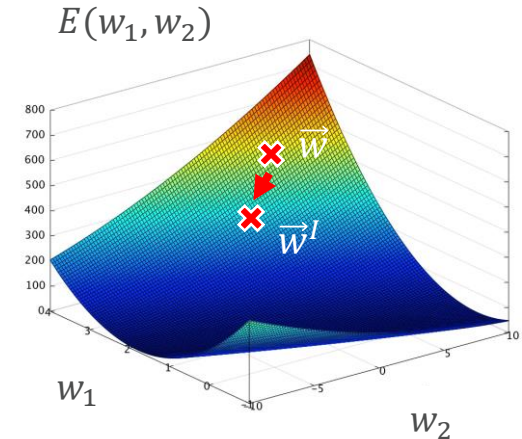


Gradient descent

- Gradient is a vector containing the partial derivatives of the cost function

$$\nabla E(w_1, \dots, w_n) = \left\langle \frac{\partial E}{\partial w_1}, \dots, \frac{\partial E}{\partial w_n} \right\rangle$$

- Using the gradient, new weights are calculated
- α step size $\rightarrow$ learning rate $\rightarrow \nabla E(\vec{w})$
- Step-by-step adjustment until gradient = 0 or smaller than threshold $\rightarrow$ convergence
- Risk of local minimum $\rightarrow$ initialization



Gradient descent

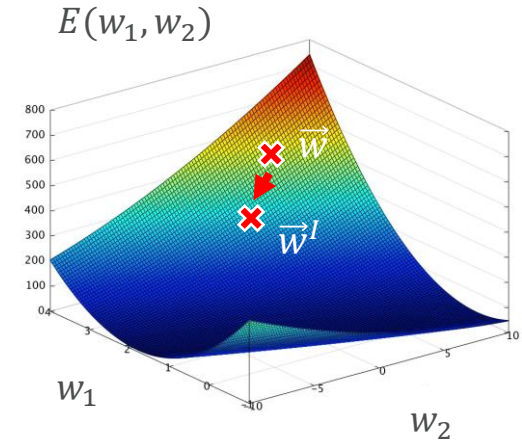
- Gradient is used to calculate new weights

$$\vec{w}^I = \vec{w} - \alpha \nabla E(\vec{w})$$

- Simplified form, actually:

$$\vec{w}^I = \vec{w} - \alpha \nabla \frac{1}{n} \sum_{x=1}^n (t_x - a_x)$$

- Consider all n training inputs during calculation of average error
- → costly, slow



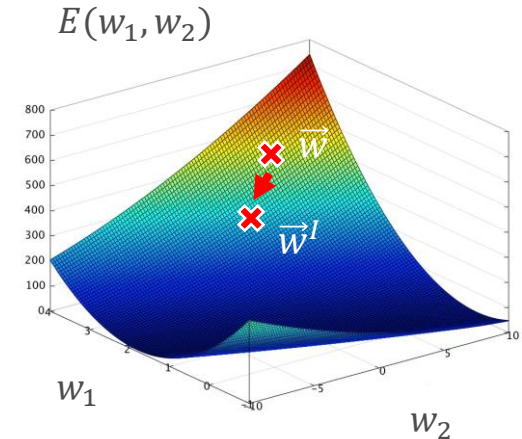
Gradient descent

$$\vec{w}^I = \vec{w} - \alpha \nabla \frac{1}{n} \sum_{x=1}^n (t_x - a_x)$$

- Stochastic Gradient Descent $\rightarrow$ approximation
- Instead of n inputs only consider one

$$\vec{w}^I = \vec{w} - \alpha \nabla (t_x - a_x)$$

- Mathematically not optimal, but “good enough”
- Path to optimum is less direct due to noise/outliers
- More iterations until convergence, BUT greatly reduce cost per iteration



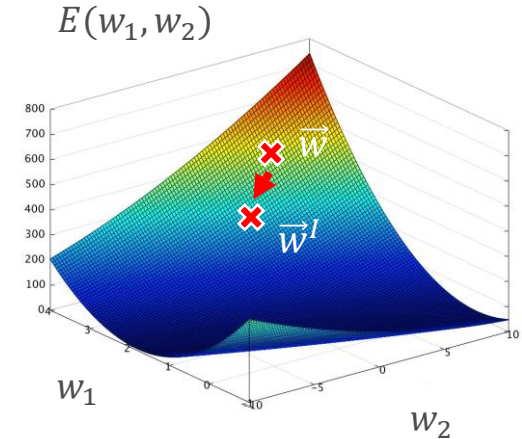
Gradient descent

$$\vec{w}^I = \vec{w} - \alpha \nabla \frac{1}{n} \sum_{x=1}^n (t_x - a_x)$$

- Mini Batch Gradient Descent → compromise
- Instead of n inputs use m with $m < n$

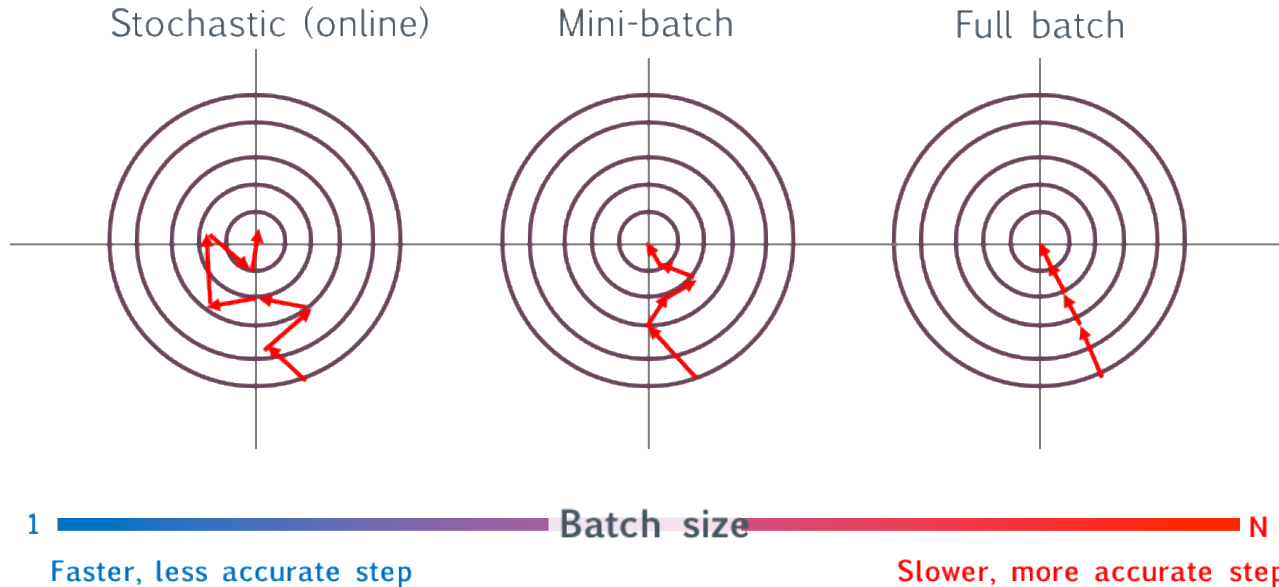
$$\vec{w}^I = \vec{w} - \alpha \nabla \frac{1}{m} \sum_{x=1}^m (t_x - a_x)$$

- “Best of both worlds”
- Most common approach for training
- Rule of thumb: $32 \leq m \leq 1024$



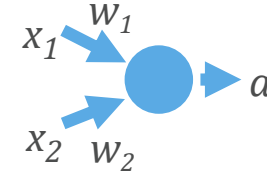
Training: Variants of Gradients

Gradient descent - approaches

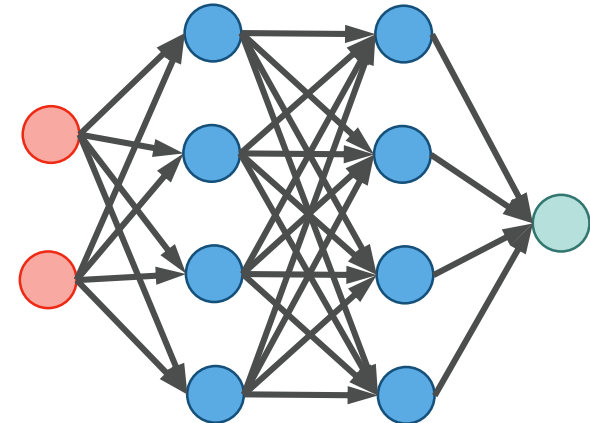


Training a multilayer network

- Core problem: indirection
- Multiple weights influence activation of multiple neurons that again influence the activation of successive neurons
- How does a certain weight influence the output of the whole network?
- Solution approach: Gradient descent allows the partial derivation of the cost function for single weights



VS.



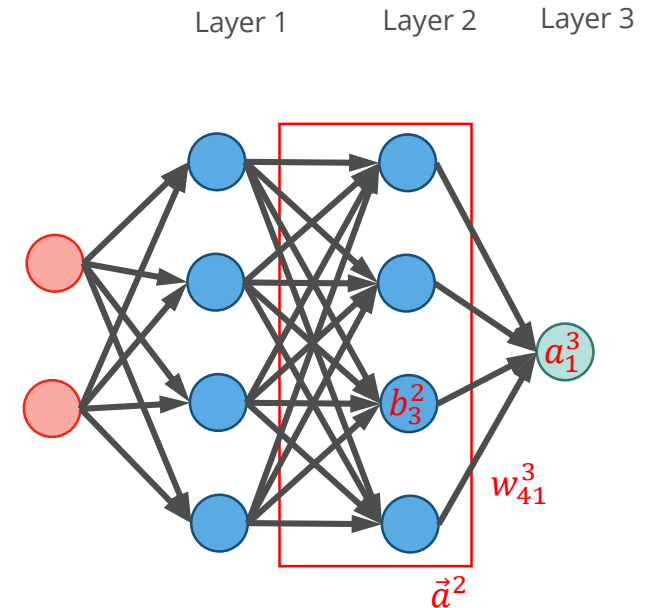
Training: Backpropagation

Training a multilayer network

- Notation:
 - w_{jk}^l weight from j-th neuron in layer (l-1) to k-th neuron in layer l
 - a_j^l, b_j^l activation/bias of j-th neuron in layer l
 - Activation of a neuron depends on the preceding layer

$$a_j^l = \sigma \left(\sum_k w_{kj}^l a_k^{l-1} + b_j^l \right)$$

$$\vec{a}^l = \sigma \left(\vec{w}^l \vec{a}^{l-1} + \vec{b}^l \right) \equiv \sigma(\vec{z}^l)$$

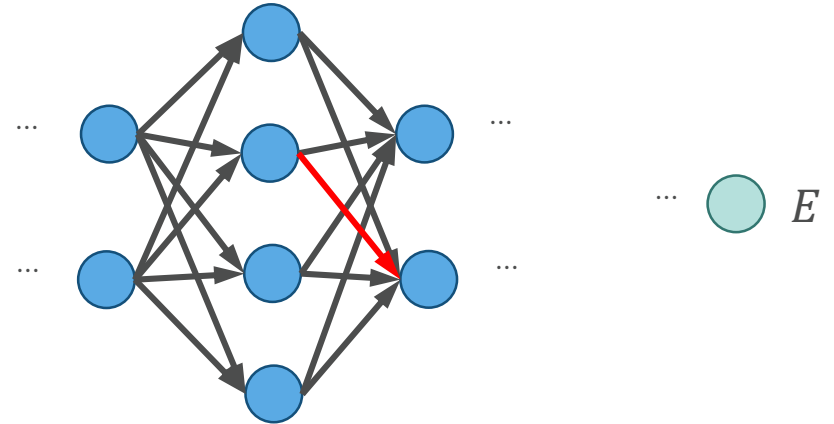


Training: Backpropagation

Backpropagation

- Understand how changes in a network influence its output

- Goal: Calculate $\frac{\partial E}{\partial w_{ij}^l}$



Training: Backpropagation

Backpropagation

- Understand how changes in a network influence its output

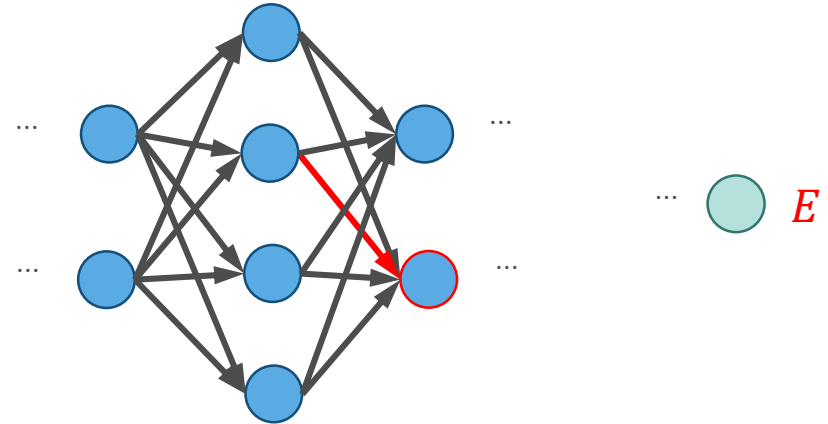
- Goal: Calculate $\frac{\partial E}{\partial w_{ij}^l}$

$$E(w_{ij}^l) = \frac{1}{2} (t - a_j^l)^2$$

$$E(w_{ij}^l) = \frac{1}{2} (t - \sigma(z_j^l))^2$$

$$E(w_{ij}^l) = \frac{1}{2} (t - \sigma(\sum_k w_{kj}^l a_k^{l-1}))^2$$

- Chain rule



Training: Backpropagation

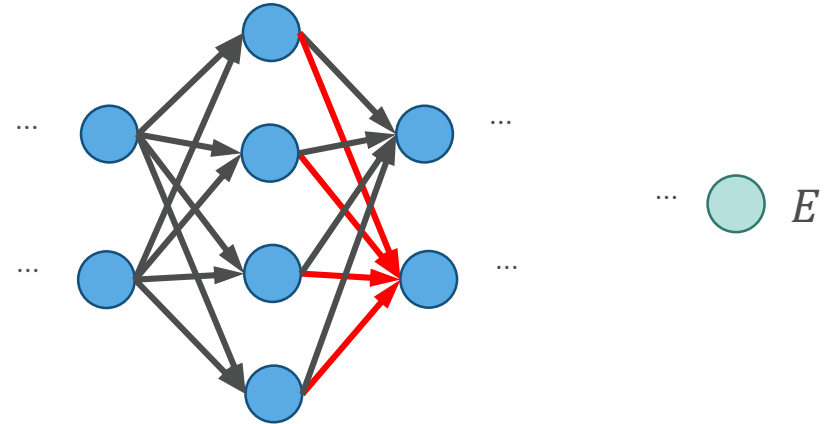
Backpropagation

- → Chain rule

$$\frac{\partial E}{\partial w_{ij}^l} = \frac{\partial E}{\partial a_j^l} \cdot \frac{\partial a_j^l}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial w_{ij}^l}$$

$$\frac{\partial z_j^l}{\partial w_{ij}^l} = \frac{\partial}{\partial w_{ij}^l} \sum_k w_{kj}^l a_k^{l-1} = \frac{\partial}{\partial w_{ij}^l} w_{ij}^l a_i^{l-1} = a_i^{l-1}$$

- In the weighted input, only one term depends on w_{ij}^l



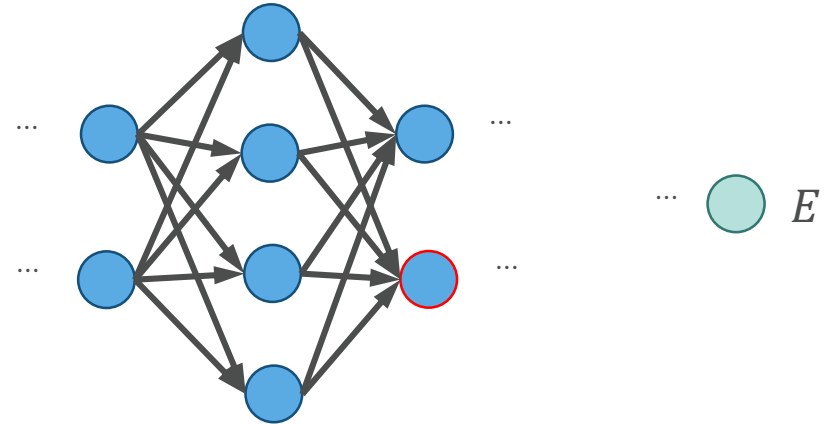
Backpropagation

- → Chain rule

$$\frac{\partial E}{\partial w_{ij}^l} = \frac{\partial E}{\partial a_j^l} \cdot \frac{\partial a_j^l}{\partial z_j^l} \cdot a_i^{l-1}$$

$$\frac{\partial a_j^l}{\partial z_j^l} = \frac{\partial}{\partial z_j^l} \sigma(z_j^l) = \sigma(z_j^l)(1 - \sigma(z_j^l))$$

- Derivative of the neuron output with regards to its input corresponds with the derivative of the activation function
- Activation functions typically offer comfortable derivatives



Training: Backpropagation

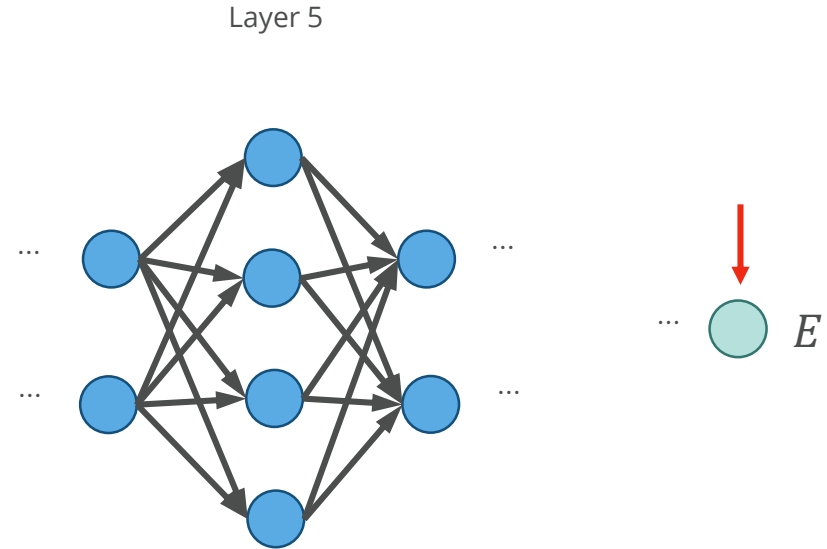
Backpropagation

- → Chain rule

$$\frac{\partial E}{\partial w_{ij}^l} = \frac{\partial E}{\partial a_j^l} \cdot \sigma(z_j^l)(1 - \sigma(z_j^l)) \cdot a_i^{l-1}$$

$$\frac{\partial E}{\partial a_j^l} = \frac{\partial}{\partial y} \frac{1}{2} (t - y)^2 = y - t$$

- Simple if neuron on the output layer → $a_j^l = y$
- Wanted value available for error calculation



- $\rightarrow$ Chain rule

- Error of neuron j
- Propagation to subsequent neurons

$$z_2^6 = \cdots + w_{32}^6(\delta_3^5) + \cdots$$



Training: Backpropagation

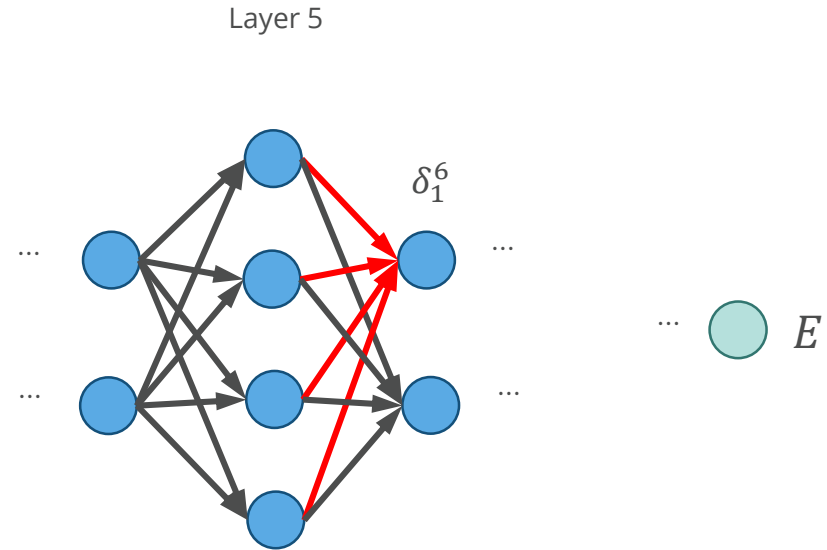
Backpropagation

- $\rightarrow$ Chain rule

$$\frac{\partial E}{\partial w_{ij}^l} = \underbrace{(y - t) \cdot \sigma(z_j^l)(1 - \sigma(z_j^l))}_{\delta_j^l} \cdot a_i^{l-1}$$

- Error of neuron j
- Propagation to subsequent neurons

$$\delta_1^6 = \sum_k w_{k1}^6 \delta_k^5 \cdot \sigma(z_1^6)(1 - \sigma(z_1^6))$$



Training: Backpropagation

Backpropagation

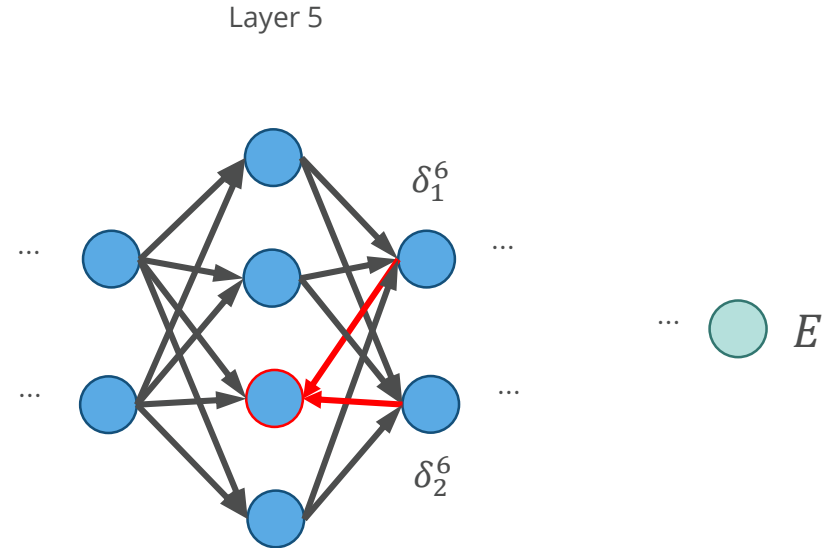
- → Chain rule

$$\frac{\partial E}{\partial w_{ij}^l} = (y - t) \cdot \underbrace{\sigma(z_j^l)(1 - \sigma(z_j^l))}_{\delta_j^l} \cdot a_i^{l-1}$$

- Error of neuron j
- backpropagation to preceding neurons

$$\delta_3^5 = (w_{31}^6 \delta_1^6 + w_{32}^6 \delta_2^6) \cdot \sigma(z_3^5)(1 - \sigma(z_3^5))$$

$$\delta_k^l = \left(\sum_{i=1}^m w_{ki}^{l+1} \delta_i^{l+1} \right) \cdot \sigma(z_k^l)(1 - \sigma(z_k^l))$$



Backpropagation - Algorithm

- For each training tuple/mini batch:
 - **Feedforward**: Calculate all weighted inputs $\vec{z}^l$ and all neuron outputs $\vec{a}^l$
 - **Output error**: Calculate errors of the output layer using

$$\delta_j = (y - t) \cdot \sigma(z_j^l)(1 - \sigma(z_j^l))$$

- **Backpropagation**: stepwise backwards propagation of the error through the network using

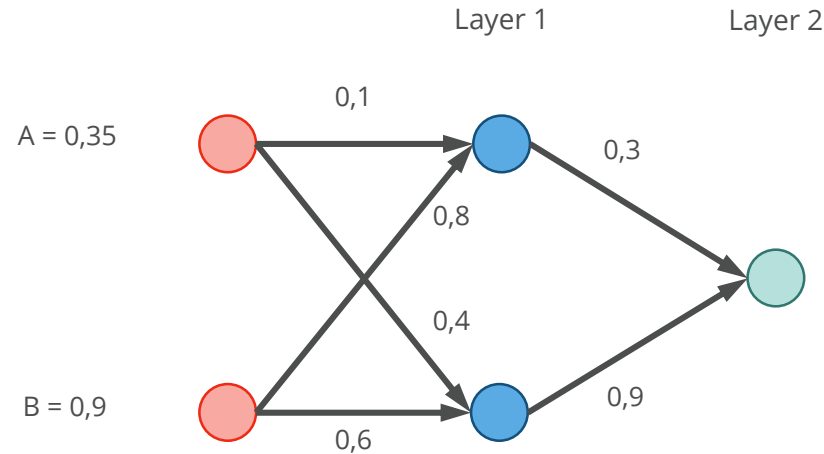
$$\delta_k^l = \left(\sum_{i=1}^m w_{ki}^{l+1} \delta_i^{l+1} \right) \cdot \sigma(z_k^l)(1 - \sigma(z_k^l))$$

- Update weights:

$$w_{ij}^l \leftarrow w_{ij}^l - \alpha \delta_j^l a_i^{l-1}$$

Training: Backpropagation

Backpropagation - Example



$$z_1^1 = (0,35 \cdot 0,1) + (0,9 \cdot 0,8) = 0,76$$

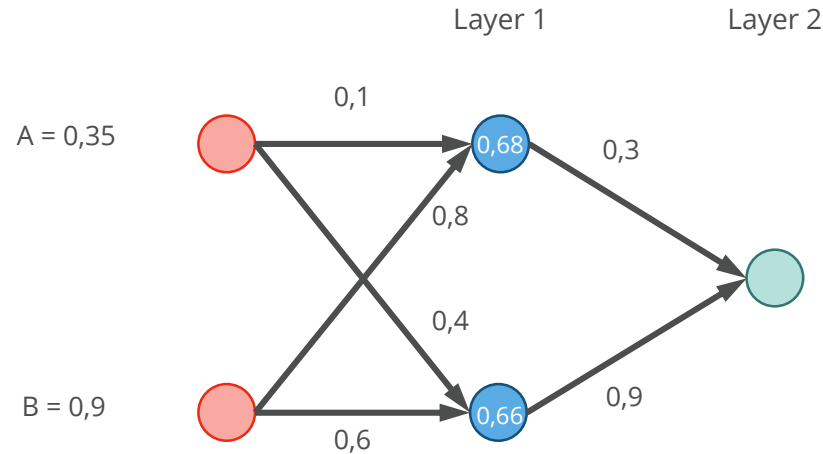
$$a_1^1 = \sigma(0,76) = 0,68$$

$$z_2^1 = (0,35 \cdot 0,4) + (0,9 \cdot 0,6) = 0,68$$

$$a_2^1 = \sigma(0,68) = 0,66$$

Training: Backpropagation

Backpropagation - Example



t = 0,5

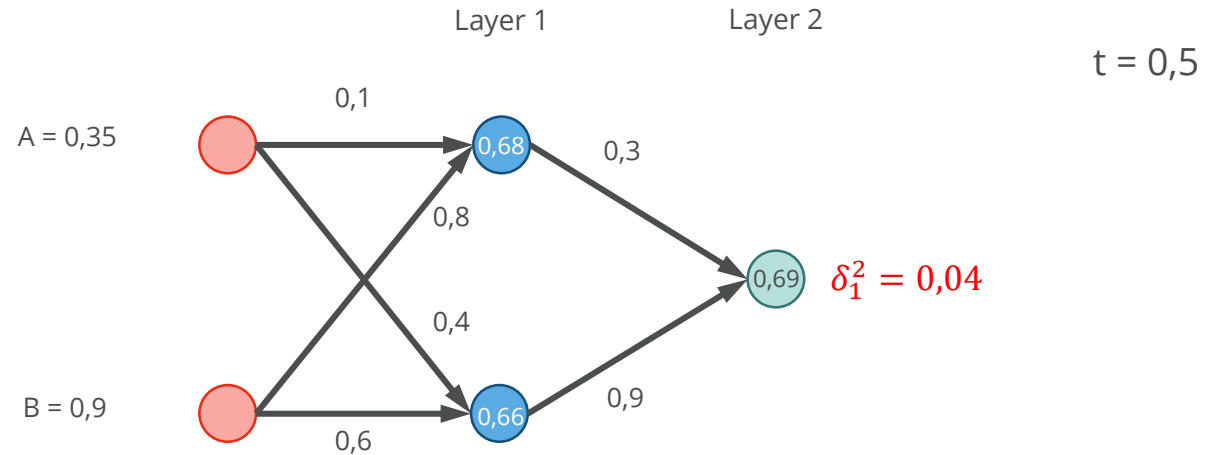
$$z_1^2 = (0,3 \cdot 0,68) + (0,9 \cdot 0,66) = 0,8$$

$$a_1^2 = \sigma(0,8) = 0,69$$

$$\delta_1^2 = (0,69 - 0,5)(0,69)(1 - 0,69) = 0,04$$

Training: Backpropagation

Backpropagation - Example

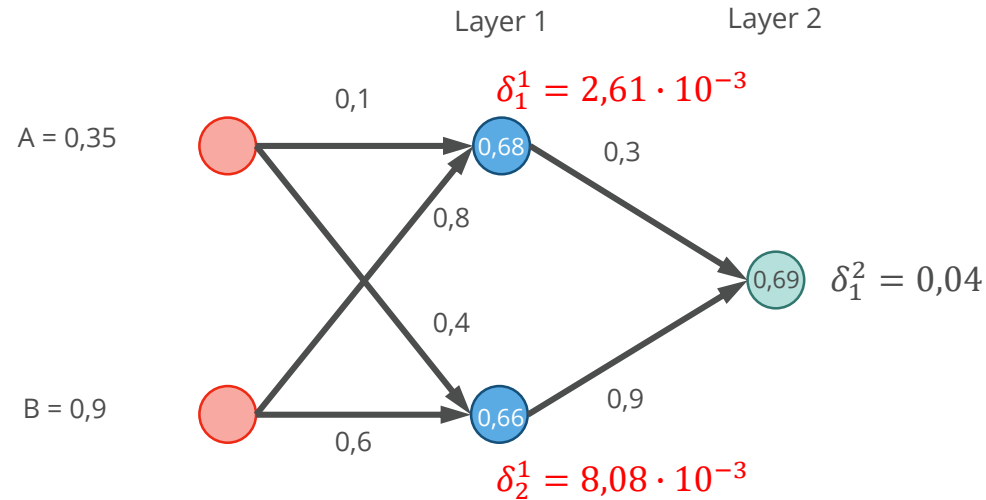


$$\delta_1^1 = (0,3 \cdot (0,04))(0,68)(1 - 0,68) = 2,61 \cdot 10^{-3}$$

$$\delta_2^1 = (0,9 \cdot (0,04))(0,66)(1 - 0,66) = 8,08 \cdot 10^{-3}$$

Training: Backpropagation

Backpropagation - Example

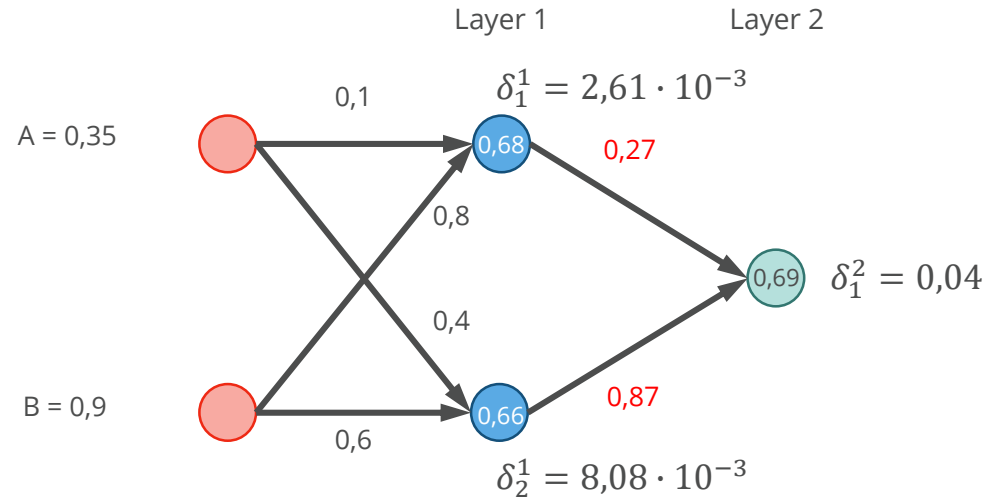


$$w_{11}^{2+} = w_{11}^2 - \alpha \delta_1^2 a_1^1 = 0,3 - (0,04 \cdot 0,68) = 0,27$$

$$w_{21}^{2+} = w_{21}^2 - \alpha \delta_1^2 a_2^1 = 0,9 - (0,04 \cdot 0,66) = 0,87$$

Training: Backpropagation

Backpropagation - Example



$t = 0,5$

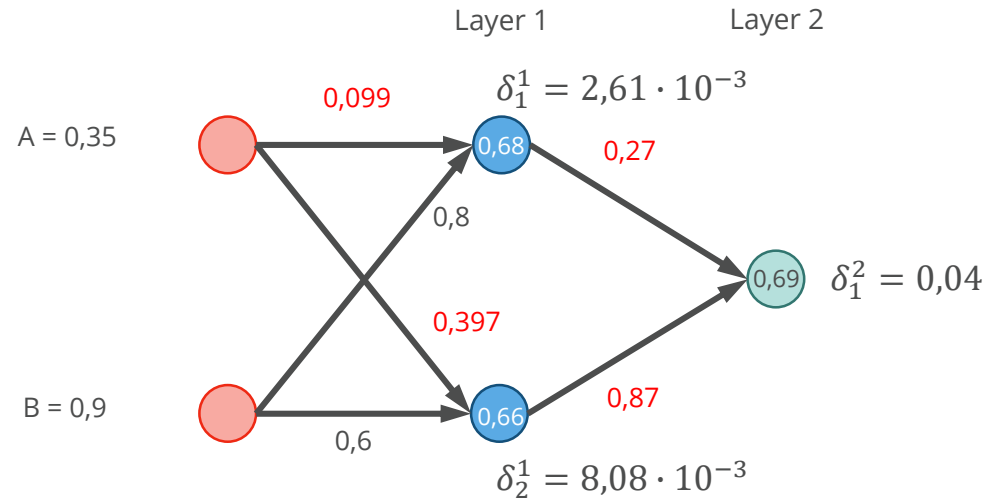
$\alpha = 1$

$$w_{11}^{1+} = w_{11}^1 - \alpha \delta_1^1 A = 0,1 - (2,61 \cdot 10^{-3} \cdot 0,35) = 0,099$$

$$w_{12}^{1+} = w_{12}^1 - \alpha \delta_2^1 A = 0,4 - (8,08 \cdot 10^{-3} \cdot 0,35) = 0,397$$

Training: Backpropagation

Backpropagation - Example



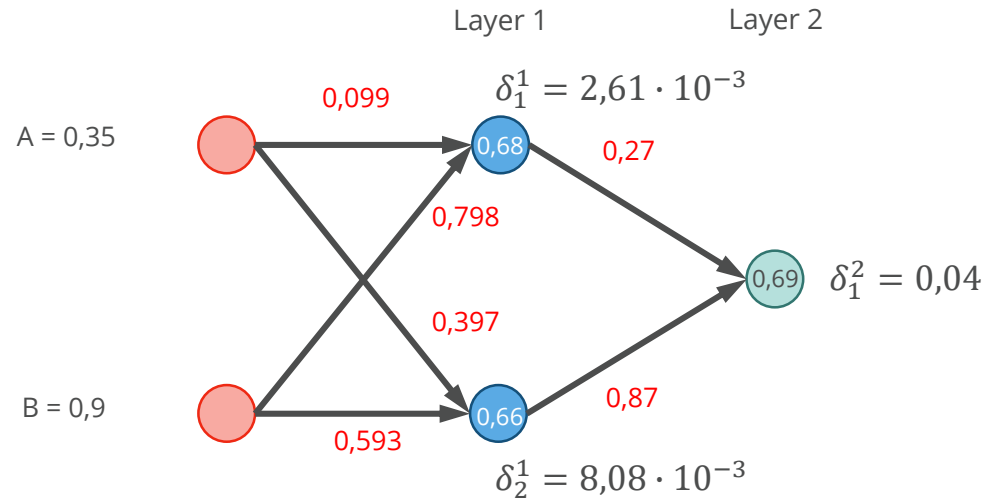
$t = 0,5$
 $\alpha = 1$

$$w_{21}^{1+} = w_{21}^1 - \alpha \delta_1^1 B = 0,8 - (2,61 \cdot 10^{-3} \cdot 0,9) = 0,798$$

$$w_{22}^{1+} = w_{22}^1 - \alpha \delta_2^1 B = 0,6 - (8,08 \cdot 10^{-3} \cdot 0,9) = 0,593$$

Training: Backpropagation

Backpropagation - Example



$$z_1^{1+} = (0,35 \cdot 0,099) + (0,9 \cdot 0,798) = 0,75$$

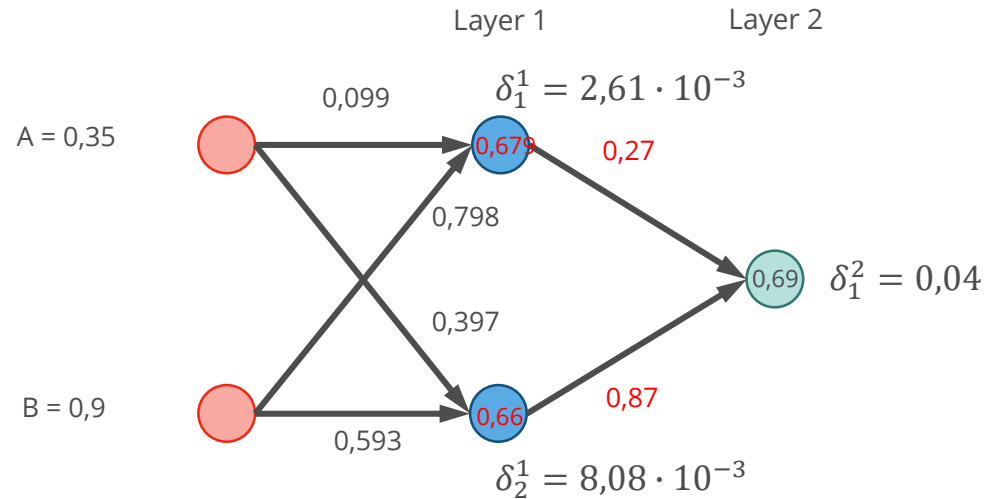
$$a_1^{1+} = \sigma(0,75) = 0,679$$

$$z_2^{1+} = (0,35 \cdot 0,397) + (0,9 \cdot 0,593) = 0,67$$

$$a_2^{1+} = \sigma(0,67) = 0,66$$

Training: Backpropagation

Backpropagation - Example



$t = 0,5$

$\alpha = 1$

$$z_1^{2+} = (0,27 \cdot 0,679) + (0,87 \cdot 0,66) = 0,76$$

$$a_1^{2+} = \sigma(0,76) = 0,68$$

$$\delta_1^{2+} = (0,68 - 0,5)(0,68)(1 - 0,68) = 0,039$$

Backpropagation – Expansions

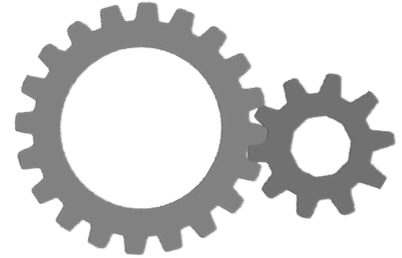
- Declining learning rate
 - The closer we get to the minimum the smaller the learning rate should be
 - Time-based decay: decrease learning rate over the course of one epoch (all training data was feedforwarded through the network)
 - Step-decay: learning rate is decreased by a fixed factor after x epochs
 - Exponential decay: exponential decrease of learning rate, see time-based
- Inertia term / Momentum
 - Weight update is influenced by the previous weight update → “ball rolling down a hill”
 - Plateaus can be crossed faster

$$\Delta w_{ij}(t + 1) = (1 - \beta)(-\alpha \delta_j a_i) + \beta \Delta w_{ij}(t)$$

Backpropagation – Problems

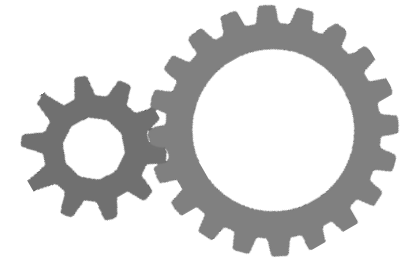
Vanishing Gradient

- Backpropagation over multiple layers leads to smaller and smaller changes $\rightarrow$ greatly decreases learning speed
- $\sigma'(x) \leq 0,25$ and $0 < w < 1$
- Repeated multiplication of small values $\rightarrow$ product gets smaller



Exploding Gradient

- $w_j \sigma'(z_j) > 1 \rightarrow$ too fast and unstable learning in the front layers
- Basic problem: gradients depend on all subsequent layers $\rightarrow$ more layers mean more instability

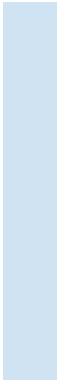


Backpropagation – Expansions

- Alternatives to gradient descent
 - Newton's method
 - Conjugate gradient method
 - Quasi-Newton method
 - Levenberg-Marquardt Algorithm (only sum-of-squared errors)
- In general these methods converge faster due to usage of Hessian and Jacobian matrices
- Calculation of these matrices requires additional storage space, therefore these methods are not suitable for large networks with thousands of parameters
- Ranking with regards to storage requirements and speed
- Gradient descent < Conjugate gradient < Newton < Quasi Newton < Levenberg-Marquardt



speed/storage

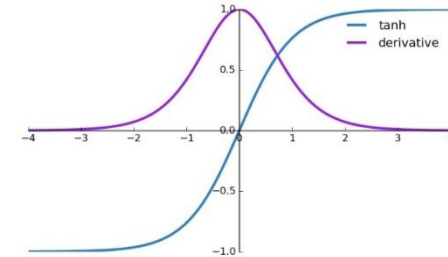
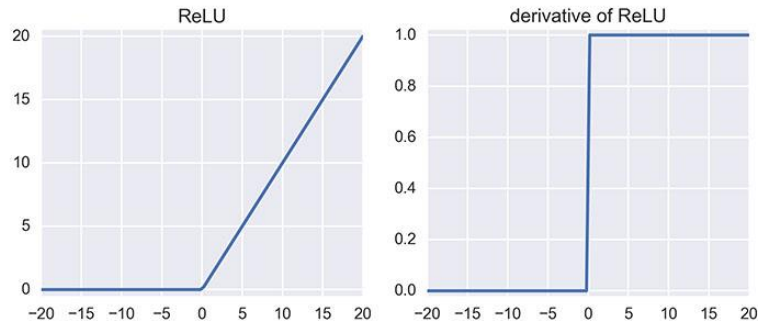
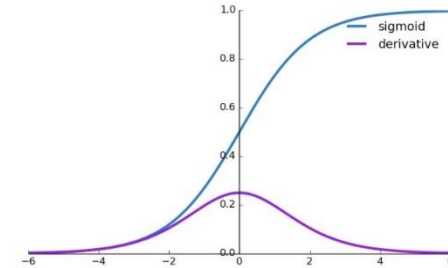


Variants of Neural Networks

Even more Activation Functions

Activation functions

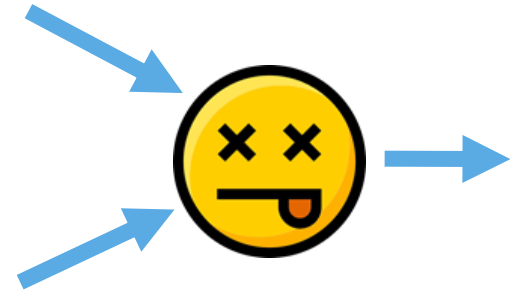
- Alternatives to sigmoid
- Tanges Hyperbolicus (tanh): scaled sigmoid
 $\tanh(z) = 2\sigma(2z) - 1, \tanh'(x) \leq 1$
- Rectified Linear Unit (ReLU): $A(z) = \max(0, z)$



Even more Activation Functions

Activation functions

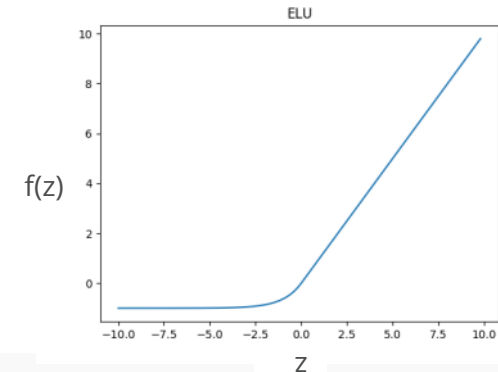
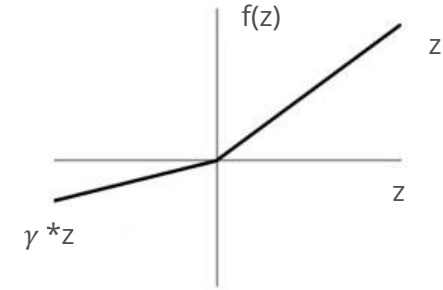
- Rectified Linear Unit (ReLU): $A(z) = \max(0, z)$
- Pros:
 - no “vanishing gradients”
 - selective neuron activation
 - easy to calculate
 - good performance
- Cons:
 - inflates activation $\rightarrow$ negative effect on learning
 - Prone to overfitting
 - “dead neuron” problem: if weights of an neuron lead to $z_j \leq 0$, the neuron can never be activated again
- $\rightarrow$ different variants



Even more Activation Functions

Activation functions

- ReLu variants
- Leaky ReLu: $LReLU(z) = \max(\gamma z, z)$ with $(\gamma < 1)$
 - Negative „mini“ gradient keeps learning alive
- Exponential Linear Unit: $ELU(\gamma, z) = \max(\gamma(e^z - 1), z)$
 - Limit for negative values
 - Brings average activation closer to zero

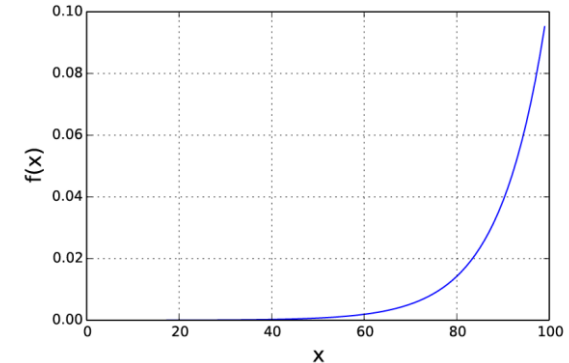


Even more Activation Functions

Activation functions

- Softmax
 - Transforms output into probabilities
 - → ideal for classification
 - Sigmoid allows only two classes
 - Generally for output layer
 - z is input vector to output layer
 - K is number of output neurons

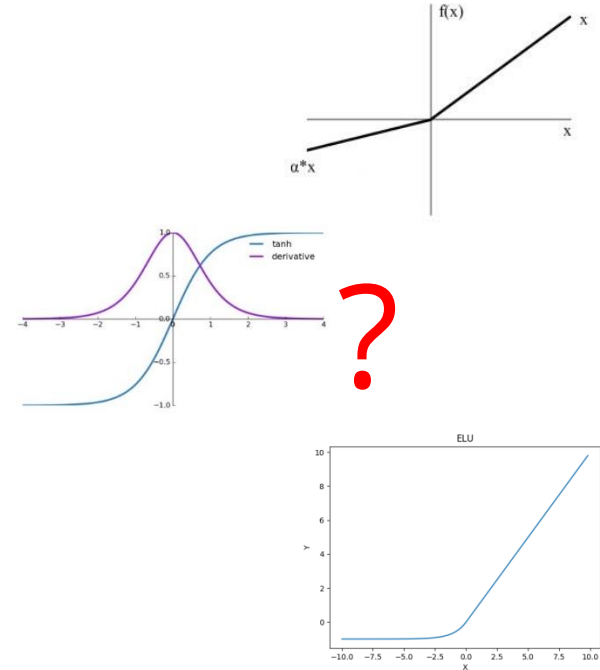
$$\text{softmax}(z)_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}$$



Even more Activation Functions

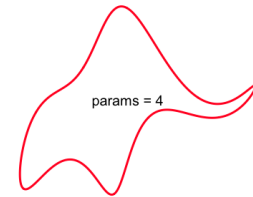
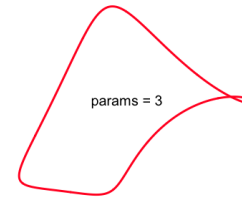
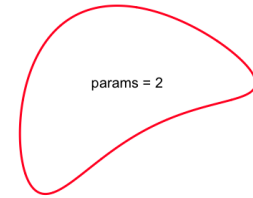
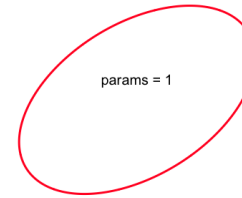
Choose the right activation functions

- What function do I use?
- “Depends on the problem”
- Sigmoid and Tangens suited for classification
- Currently ReLu is considered the best default
- However: Try!
- Why use activation functions at all?
 - Provide non-linearity → make hidden layers useful
 - Linear function of linear functions is linear



Training – Overfitting

- Enrico Fermi: „I remember my friend Johnny von Neumann used to say, with four parameters I can fit an elephant, and with five I can make him wiggle his trunk.“
- With enough parameters, everything can be modelled
- Such models are not necessarily good → Fail to generalize
- Huge problem for NNs → shitloads of parameters possible
- Solutions: more training data, smaller networks, regularization



Training – Regularization

- L2 regularization (weight decay)
- Add penalty term to cost function

$$E = \frac{1}{2n}(t - y)^2 + \frac{\lambda}{2n} \sum_w w^2$$

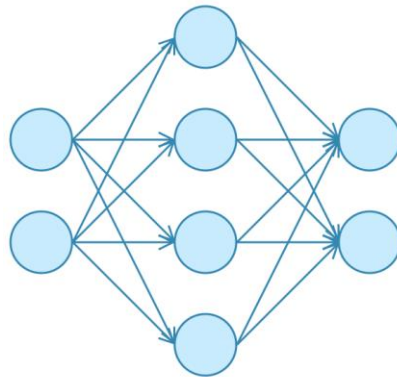
$\lambda > 0$, n number of training samples

- Favors small weights, big weights have to provide strong improvement
- Trade-off between error minimization and small weights
- Controlled using λ , large $\lambda \rightarrow$ small weights, small $\lambda \rightarrow$ error minimization
- L1 regularization:

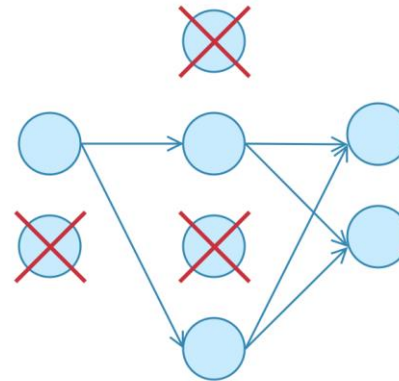
$$E = \frac{1}{2n}(t - y)^2 + \frac{\lambda}{n} \sum_w |w|$$

Training – Dropout

- Currently most popular method to prevent overfitting
- In each iteration of trainings, some neurons are “dropped”
- Inputs and output are ignored for these neurons
- Dropout rate typically 20-50%

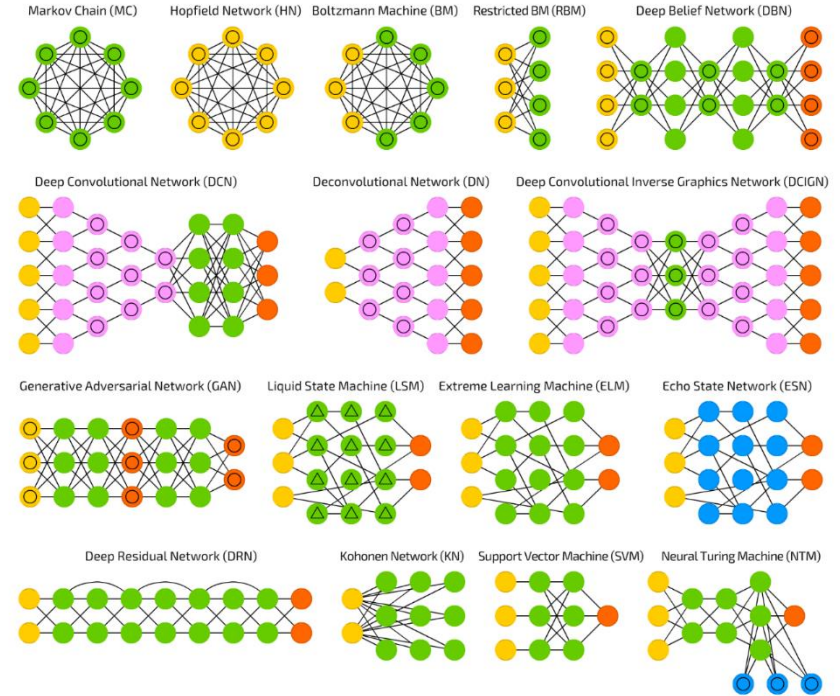
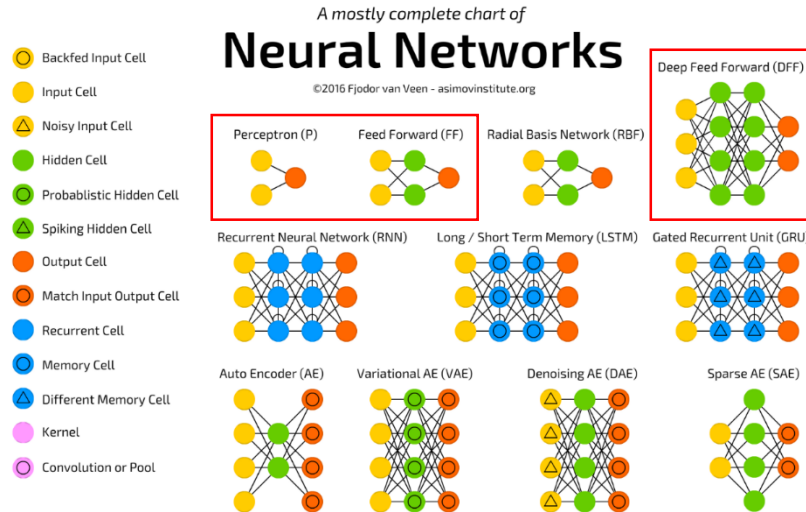


No Dropout



With Dropout

Architecture overview



Neural Networks

Scalable Data Engineering